

# Middle Square Weyl Sequence RNG

Bernard Widynski<sup>1</sup>

April 4, 2017

## Abstract

In this article, we propose a new implementation of John von Neumann's middle square random number generator (RNG). A Weyl sequence is utilized to keep the generator running through a long period.

## 1 Introduction

Early in the field of computer science, John von Neumann proposed the middle square method for generating random numbers [4]. A number is squared and the middle digits are returned as the next random number. This method was used as a source of data in the early work on Monte Carlo. Good data was produced but the middle square had a known problem which was referred to as the “zero mechanism”. Once the middle digits became zero the generator would continue to produce zero outputs. Repeating cycles of non-zero data could also be produced. In this article we propose the use of a Weyl sequence to run the middle square. The Weyl sequence overcomes the “zero mechanism”. It keeps the generator running through a long period.

## 2 Weyl Sequence

We use a Weyl sequence similar to the Weyl sequence in Brent's xorgens RNG [1]. This is actually an integer stepping sequence of period  $2^{64}$ . In C, it has the following form:  $w += s$ ; where  $w$  is an unsigned 64-bit integer and  $s$  is an odd constant. A middle square RNG run by this sequence is shown below:

```
uint64_t x = 0, w = 0, s = 0xb5ad4eceda1ce2a9;
inline static uint32_t msws() {
    x *= x; x += (w += s); return x = (x>>32) | (x<<32);
}
```

The Weyl sequence is added to the square of  $x$ . The middle is extracted by shifting right 32 bits.

### 3 Discussion

One might ask “Where is the middle?” in the above program. In C, arithmetic on unsigned 64-bit integers is modulo  $2^{64}$ . After squaring, the  $x$  value will have only the lower 64 bits of the result. These 64 bits represent the lower half of the square. Shifting right 32 bits will return the upper part of this lower half. This is the 32-bit middle that is returned.

The Weyl sequence  $w$  is non-zero for all but one point. When the middle bits of  $x$  become zero, which will happen at random, this non-zero  $w$  will be added to restart the middle square. This prevents the “zero mechanism”.

For any odd constant  $s$ , the period of  $w$  will be  $2^{64}$ . Since  $x$  is driven by  $w$ , the period of  $x$  will be at least the period of  $w$ . That is, the period of the generator will be at least  $2^{64}$  for any given odd constant.

In addition to providing a long period, the Weyl sequence also provides a basis for uniformity in the output. The parameter  $w$  is uniform and is independent of  $x$ . While  $x$  may not be uniform,  $x + w$  will be uniform. So, the output of the RNG will be uniform.

The Weyl sequence also provides the capability to generate different streams of data. This is accomplished by simply setting the constant  $s$  to a different value. This will result in a different stream of data, distinct from and non-overlapping with any other stream.

The constant  $s$  should be non-zero in the upper 32 bits and 1 in the least significant bit. The number of different  $s$  values of this type is on the order of  $2^{63}$ . Given that the stream length is  $2^{64}$ , this will provide about  $2^{127}$  random numbers in total.

One might think that non-overlapping data would always be obtained if the generator was started with different  $x$  values but the same  $w$  and  $s$  values. This is not always the case. It is possible for two different  $x$  values to have the same result mod  $2^{64}$  after squaring. Should this happen, exactly the same data would be produced even though the initial  $x$  values were different. To guarantee distinct data through the period of  $2^{64}$ , one should use a different  $s$  value for each run. This will produce different data and prevent any occurrence of overlapping data.

When generating seeds for parallel usage, one might simply randomize the  $s$  value with a 64-bit random number, verify that the upper 32 bits are non-zero, and set the least significant bit to 1. This will with great probability produce all different seeds. If it is necessary to guarantee that all are different, one might pre-compute a set of random  $s$  values and verify that all were different and then use this set when creating streams.

## 4 Statistical and Timing Results

The msrs generator passes all the tests in the stringent BigCrush battery in TestU01 [2]. A square is nonlinear and for this reason the data is likely to be quite superior to data produced by any linear based RNG. This generator passes the tests of Linear Complexity and Matrix Rank.

The time to generate one billion random numbers was computed using an Intel Core i7-4770 3.4 GHz processor running Cygwin64. To provide a basis of comparison, we also timed Marsaglia's xorwow [3], one of the fastest RNGs in current usage. The time for msrs was 1.347 seconds. The time for xorwow was 1.766 seconds.

## 5 Summary

The middle square was invented at the very beginning of computer science. Modern 64-bit computing architecture has made it possible to create a usable version which has a sufficiently long period ( $2^{64}$  per stream). Processing speed is comparable with the fastest RNGs. An easy to use stream capability makes this generator quite suitable for parallel processing. A square is nonlinear and gives this generator an advantage in quality of data over linear based generators.

A free software package is available at <http://msrsrng.wixsite.com/rand>

**Acknowledgements** The author would like to thank the Wikipedia organization without which this would not have been possible. The author would also like to thank the anonymous ACM reviewers for their constructive inputs during this process. The author is particularly thankful for R. P. Brent's suggestion to add the Weyl sequence after squaring instead of prior to squaring. This provides a basis for uniformity in the output.

## References

- [1] R. P. Brent. "Some Long-Period Random Number Generators using Shifts and Xors". In: *ANZIAM Journal* 48 (2006), pp. C188–C202. URL: <http://maths-people.anu.edu.au/~brent/pub/pub224.html> (cit. on p. 1).
- [2] P. L'Ecuyer and R. Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software* 33 (2007). URL: <http://simul.iro.umontreal.ca/testu01/tu01.html> (cit. on p. 3).

- [3] George Marsaglia. “Xorshift RNGs”. In: *Journal of Statistical Software* 8.14 (2003). URL: [www.jstatsoft.org/v08/i14/paper](http://www.jstatsoft.org/v08/i14/paper) (cit. on p. 3).
- [4] John von Neumann. “Various Techniques Used in Connection with Random Digits”. In: *A.S. Householder, G.E. Forsythe, and H.H. Germond, eds., Monte Carlo Method, National Bureau of Standards Applied Mathematics Series 12* (1951), pp. 36–38 (cit. on p. 1).

### Author address

1. **Bernard Widynski**,  
<mailto:mwsrnrng@gmail.com>